Modeling multi-clocked data-flow programs in the Generic Modeling Environment

Loïc Besnard, Christian Brunette, Thierry Gautier, Jean-Pierre Talpin

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1771

MODELING MULTI-CLOCKED DATA-FLOW PROGRAMS IN THE GENERIC MODELING ENVIRONMENT

LOÏC BESNARD , CHRISTIAN BRUNETTE , THIERRY GAUTIER , JEAN-PIERRE TALPIN

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Modeling multi-clocked data-flow programs in the Generic Modeling Environment

Loïc Besnard[*] , Christian Brunette[**] , Thierry Gautier[***] , Jean-Pierre Talpin[****]

Systèmes communicants
Projet ESPRESSO

Publication interne n ° 1771 — December 2005 — 40 pages

Abstract: This paper presents Signal-Meta, the metamodel designed for the synchronous data-flow language SIGNAL. It relies on the Generic Modeling Environment (GME), a configurable object-oriented toolkit that supports the creation of domain-specific modeling and program synthesis environments. The graphical description constitutes the base to build environments to design multi-clocked systems, and a good front-end for the POLYCHRONY platform. To complete this front-end, we develop a tool that transforms the graphical Signal-Meta specifications to the corresponding SIGNAL program.

Key-words: Metamodeling, GME, synchronous languages, SIGNAL

(Résumé : tsvp)

[*] loic.besnard@irisa.fr
[**] christian.brunette@irisa.fr
[***] thierry.gautier@irisa.fr
[****] jean-pierre.talpin@irisa.fr

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes
Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

Modélisation de programmes orientés flot de données multi-horloges dans GME

Résumé : Cet article présente Signal-Meta, le métamodèle conu pour le langage synchrone orienté flot de données SIGNAL. Signal-Meta est basé sur l'environnement générique de modélisation orienté objet GME. GME fournit un ensemble configurable d'outils permettant la création d'environnements aussi bien de modélisation pour des domaines spécifiques que de synthèse de programmes. Signal-Meta constitue une brique de base pour construire des environnements pour modéliser des systèmes multi-horloges et un bon éditeur graphique de modélisation pour POLYCHRONY. Pour réaliser la liaison entre Signal-Meta et POLYCHRONY, nous avons développé un outil transformant les spécifications graphiques de Signal-Meta en code SIGNAL.

Mots clés : Métamodélisation, GME, langages synchrones, SIGNAL

Contents

1 Introduction

The *synchronous hypothesis* has been proposed in the late '80s and extensively used ever since to facilitate design of control-dominated systems. Nowadays synchronous languages are commonly used in the European industry, especially in avionics, to rapidly prototype, simulate, verify and synthesize embedded software for mission critical applications. However, synchronous programming languages, such as Lustre, Lucid, Esterel, Signal are most commonly regarded as "domain-specific" languages, as their usage is mostly restricted to aid highly-trained engineers to design mission-critical systems.

In the aim of bringing synchronous technologies to a vaster community aware of model-driven engineering, we have developed a simple and highly extensible interface to the Polychrony workbench, that implements the multi-clocked synchronous data-flow language Signal, with the Generic Modeling Environment (or Gme) [13]. This interface is the medium to experiments with relating the polychronous model of computation of the workbench with more ergonomic and diagrammatic notations such as data-flow diagrams, Uml state diagrams and the combination of both as mode automata. The aim of this experiment is to find the simplest and most ergonomic representation of a formal model of time such as that of the Polychrony workbench in the forthcoming real-time profiles for more vastly known notations such as the Uml state diagrams.

The remainder is organized as follows. Sections 2 and 3 first introduce respectively the Signal language and the Generic Modeling Environment. Section 4 describes Signal-Meta, the metamodel of Signal specified in Gme. Section 5 illustrates how to use this metamodel through the description of an example. Section 6 presents the component added to Gme to transform the graphical specifications into a Signal code. The adopted approach is discussed in Section 7 and finally, conclusions and future works are given in Section 8.

2 POLYCHRONY and SIGNAL

Among other synchronous frameworks, the Polychrony workbench, available from [9], implements an original model of time as *partially ordered* synchronization and scheduling relations, to provide the ability to model high-level abstractions of systems paced by multiple clocks: locally synchronous and globally asynchronous systems. It provides a flexible way to model heterogeneous and complex distributed embedded systems at a high level of abstraction, while reasoning within a simple and formally defined mathematical model.

In Polychrony, design proceeds in a compositional and refinement-based manner by first considering a weakly timed data-flow model of the system under consideration and then provides expressive timing relations to gradually refine its synchronization and scheduling structure to finally check correctness of the assembled components using assumption/guarantee reasoning. The synchronous language Signal, which is associated with Polychrony, favors the progressive design of correct by construction systems by means of well-defined model transformations, that preserve the intended semantics of early require-

ment specifications to eventually provide a functionally correct deployment on the target architecture.

The POLYCHRONY IDE offers several tools including the SIGNAL batch compiler that provides a set of functionalities, such as program transformations, optimizations, formal verification, and code generation. POLYCHRONY includes the SIGALI model checker [14], which enables both verification and controller synthesis, and it also includes a graphical user interface for SIGNAL.

The following presents first the syntax of the SIGNAL language in Section 2.1, and in Section 2.2, the multi-clocked data-flow graphs, which are the internal representation used for analysis and transformation of programs. The semantics of multi-clocked data-flow graphs is described by considering the theory of synchronous micro-step automata proposed by Potop et al. in [16] (see also the micro automata defined for the DC+ format in [7] and for SIGNAL in [3]). The general framework of micro-step synchronous automata is presented in Section 2.3, and an operational semantics of SIGNAL's data-flow graphs is described in Section 2.4.

2.1 Syntax

The SIGNAL language handles unbounded series of typed values $(x_t)_{t\in\mathbb{N}}$, called *signals*, denoted as `x` and implicitly indexed by discrete time. At a given instant, a signal may be present, at which point it holds a value; or absent. The set of instants where a signal `x` is present is called its *clock*. It is noted as `^x`. Signals that have the same clock are said to be *synchronous*. A SIGNAL *process* is a system of equations over signals that specifies relations between values and clocks of the involved signals. A *program* is a process. SIGNAL relies on a six primitive constructs, which are combined using a composition operator:

- An equation `y:= f(x)` describes a relation between a sequence of operands `x` and a sequence of results `y` by a process `f`.
- A delay equation `x := y$n init v` initially defines the signal `x` by the value `v` and then by the n-th previous value of the signal `y`. In a delay equation, the signals `x` and `y` are assumed to be synchronous, i.e., either simultaneously present or simultaneously absent at all times.
- A sampling `x := y when z` defines `x` by `y` when `z` is true and both `y` and `z` are present. In a sampling equation, the output signal `x` is present iff both input signals `y` and `z` are present and `z` holds the value *true*.
- A merge `x := y default z` defines `x` by `y` when `y` is present and by `z` otherwise. In a merge equation, the output signal is present iff either of the input signals `y` or `z` is present.
- The synchronous composition `(| P | Q |)` of the processes `P` and `Q` consists of simultaneously considering a solution of the equations in `P` and `P` at any time.

- The hiding equation `P where x` restricts the lexical scope of a signal `x` to a process `P`.

These primitives are of sufficient expressive power to derive other constructs for comfort and structuring: the clock synchronization operator (`^=`) for example. The equation `x ^= y` synchronizes the clocks of signals `x` and `y`. It corresponds using SIGNAL's primitives to `(| h := (^x = ^y) |) where h`.

SIGNAL provides a process model in which any SIGNAL process may be "encapsulated" (see an example in FIG. 5). Different categories of process models are syntactically distinguished: these are actions, functions, nodes, and processes. This process frame allows to abstract a process to an interface, so that the process can be used afterwards as a black box through its interface. This interface describes parameters, input-output signals and clock and dependence relations between them. A process model also enables the definition of sub-processes. Sub-processes that are specified by an interface without any internal behavior are considered as external (they may be separately compiled processes or physical components). On the other hand, SIGNAL allows to import external modules (e.g. C++ functions). Finally, put together, all these features of the language favor modularity and re-usability.

2.2 SIGNAL's data-flow graph

The data-flow synchronous formalism SIGNAL supports an intermediate representation of multi-clocked specification that exposes its control and data-flow properties for the purpose of analysis and transformation. A process p is represented as a data-flow graph G. In this graph, a vertex g is a data-flow relation that partially defines a clock or a signal. A signal vertex $c \Rightarrow x = f(y_{1..n})$ partially defines x by $f(y_{1..n})$ at the clock c. A clock vertex $\hat{x} = e$ defines a relation between two particular signals or events called clocks.

$$\begin{array}{rl} G, H ::= g \,|\, (G \,|\, H) \,|\, G \text{ where } x & \text{(graph)} \\ g, h ::= \hat{x} = e \,|\, c \Rightarrow x = f(y_{1..n}) & \text{(vertices)} \end{array}$$

A clock c expresses control and defines a condition upon which a data-flow relation is executed. The clock $\hat{x}$ defines when the signal x is present (its value is available). The clocks x and $\neg x$ mean that x is respectively true and false, and hence present. A clock expression e is a boolean expression that defines how a clock is computed. 0 means never, and the operators $\hat{+}$, $\hat{*}$, and $\hat{-}$ correspond respectively to the union, intersection, and complementary of clocks.

$$c ::= \hat{x} \,|\, x \,|\, \neg x \quad \text{(clock)} \qquad e ::= 0 \,|\, c \,|\, e_1 \hat{-} e_2, \,|\, e_1 \hat{+} e_2 \,|\, e_1 \hat{*} e_2 \quad \text{(expression)}$$

The decomposition of a process into the synchronous composition of clock and signal vertices is defined by induction on the structure of p. Each equation is decomposed into data-flow functions guarded by a condition, the clock $\hat{x}$ of the output. This clock will need to be

computed for the function to be executed.

$$
\begin{aligned}
G_{[x=y\$1 \text{ init } v]} &\stackrel{\text{def}}{=} (\hat{x} \Rightarrow x = y\$1 \text{ init } v) \,|\, (\hat{x} = \hat{y}) \\
G_{[x=y \text{ when } z]} &\stackrel{\text{def}}{=} (\hat{x} \Rightarrow x = y) \,|\, (\hat{x} = \hat{y} \,\hat{}* z) \\
G_{[x=y \text{ default } z]} &\stackrel{\text{def}}{=} (\hat{y} \Rightarrow x = y) \,|\, (\hat{z} \,\hat{}- \hat{y} \Rightarrow x = z) \,|\, (\hat{x} = \hat{y} \,\hat{}+ \hat{z}) \\
G_{[p \,|\, q]} &\stackrel{\text{def}}{=} G_{[p]} \,|\, G_{[q]} \\
G_{[p \text{ where } x]} &\stackrel{\text{def}}{=} G_{[p]} \text{ where } x
\end{aligned}
$$

2.3 Micro-step synchronous automata

Micro-step automata communicate through signals $x \in X$. The *labels* $l \in L_X$ generated by the set of names X are represented by a partial map of *domain* from a set of signals X noted $\text{vars}(l)$ to a set of values $V^\perp = V \cup \{\perp\}$. The label $\perp$ denotes the *absence* of communication during a transition of the automaton. We note $l' \leq l$ iff there exists l'' disjoint from l' such that $l = l' \cup l''$ and then $l \setminus l' = l''$. We say that l and l' are *compatible*, written $l \bowtie l'$, iff $l(x) = l'(x)$ for all $x \in \text{vars}(l) \cap \text{vars}(l')$ and, if so, note $l \cup l'$ their union. We write $\text{supp}(l) = \{x \in X \,|\, l(x) \neq \perp\}$ for the *support* of a label l and $\perp_X$ for the empty support.

Synchronous automata account for primitive communications using read and write operations on *directed communication channels* pairing variables x with directions represented by tags. Emitting a value v along a channel x is written $!x = v$ and receiving it $?x = v$. We write $\text{vars}(D)$ for the channel names associated to a set of directed channels D. The undirected or untagged variables of a synchronous automaton are its *clocks* noted c.

An *automaton* $A = (s^0, S, X, \rightarrow)$ is defined by an initial state s^0, a finite set of states S noted s or $x = v$, labels L_X and by a transition relation $\rightarrow$ on $S \times L_X \times S$. The *product* $A_1 \otimes A_2$ of $A_i = (s_i^0, S_i, X_i, \rightarrow_i)$ for $0 < i \leq 2$ is defined by $((s_1^0, s_2^0), S_1 \times S_2, X_1 \cup X_2, \rightarrow)$ where $(s_1, s_2) \rightarrow^l (s_1', s_2')$ iff $s_i \rightarrow^{l|_{X_i}} s_i'$ for $0 < i \leq 2$ and $l|_{X_i}$ the projection of l on X_i. An automaton $A = (s^0, S, X, \rightarrow)$ is *concurrent* iff $s \rightarrow^\perp s$ for all $s \in S$ and if $s \rightarrow^l s'$ and $l' \leq l$ then there exists $s'' \in S$ such that $s \rightarrow^{l'} s''$ and $s'' \rightarrow^{l \setminus l'} s'$. A *synchronous* automaton $A = (s^0, S, X, c, \rightarrow)$, of clock $c \in X$, consists of a concurrent automaton $(s^0, S, X, \rightarrow)$ which satisfies

1. $s \rightarrow^l s$ implies $l = c$ or $c \not\leq l$
2. $s^0 \rightarrow^c s^0$
3. $s \rightarrow^c s'$ implies $s' \rightarrow^c s'$
4. $s_{k-1} \rightarrow^{l_k} s_k$ and $l_k \neq c \forall k \in]0, n]$
   then $\forall i, j \in]0, n]$ and $i \neq j, \text{vars}(l_i) \cap \text{vars}(l_j) = \emptyset$.

We assume that a channel x connects at most one emitter with at most one receiver. Multicast will however be used in examples and is modeled by substituting variable names (one $!x = v$ and two $?x = w_{1,2}$ will be substituted by two $!x = v$, $!x_2 = v$ and two $?x = w_1$ $?x_2 = w_2$ by introducing a local signal x_2).

The composition of automata is defined by synchronized product and synchronous communication using 1-place synchronous FIFO buffers. The synchronous FIFO of clock c and channel x is noted $\textsc{sfifo}(x,c)$. It serializes the emission event $!x = v$ followed by the receipt event $?x = v$ within the same transition (the clock tick c occurs after).

$$\textsc{sfifo}(x,c) \stackrel{\text{def}}{=} \left(s_0, \{s_{0..2}\}, \{?x, !x, c\}, c, \; c \circlearrowright s_0 \xrightarrow{!x=v} s_1 \xrightarrow{?x=v} s_2 \; \text{with } s_2 \xrightarrow{c} s_0 \right)$$

Two synchronous automata are *composable* if their tagged variables are mutually disjoint. Let $A_i = (s_i^0, S_i, X_i, c_i, \rightarrow_i)_{i=1,2}$ be two composable synchronous automata and c a clock and write $A[c_2/c_1]$ for the substitution of c_1 by c_2 in A. The synchronous composition $A_1 \,|^c A_2$ is defined by the product of A_1, of A_2 and of a series of synchronous FIFO buffers $\textsc{sfifo}(x,c)$ that are all synchronized at the same clock c.

$$A_1 \,|^c A_2 \stackrel{\text{def}}{=} (A_1[c/c_1]) \otimes \left(\bigotimes_{x \in\, (\text{vars}(X_1) \cap \text{vars}(X_2))} \textsc{sfifo}(x,c) \right) \otimes (A_2[c/c_2])$$

2.4 Micro-step semantics of SIGNAL's data-flow graphs

As already demonstrated for SIGNAL in [17], this framework accurately renders concurrency and causality for synchronous (multi-clocked) specifications. In the following, before presenting these semantics, we first introduce the SIGNAL syntax and some of their constructs.

Clocks A clock expression e corresponds to a transition system T_e^{st} from s to t which evaluates the presence of signals in accordance to e.

$$T_c^{s,t} \stackrel{\text{def}}{=} \left(s \xrightarrow{l_c} t \right) \qquad T_{c \wedge d}^{s,t} \stackrel{\text{def}}{=} \left(\begin{array}{ccccc} & \overset{l_c}{\nearrow} & s' & \overset{l_d}{\searrow} & \\ s & & \xrightarrow{l_c l_d} & & t \\ & \underset{l_d}{\searrow} & t' & \underset{l_c}{\nearrow} & \end{array} \right) / s't' \qquad T_{c \vee d}^{s,t} \stackrel{\text{def}}{=} \left(T_{c \wedge d}^{st} \cup T_c^{st} \cup T_d^{st} \right)$$

We write l_c for the label l that corresponds to the clock c and canonically denote v_x the generic value of the signal x.

$$l_{\hat{x}} \stackrel{\text{def}}{=} (?x = v_x) \qquad l_x \stackrel{\text{def}}{=} (?x = 1) \qquad l_{\neg x} \stackrel{\text{def}}{=} (?x = 0)$$

Relations A synchronization relation $\hat{x} = e$ accepts the events $\hat{x}$ and e in any order, or none of them, and then performs a clock transition c. Hence, the conditions expressed by $\hat{x}$ and e need to occur at the same time.

$$A_{\hat{x}=e} \stackrel{\text{def}}{=} \left(s, \{s,t\}, \{c, x\} \cup \text{vars}(e), c, \left(t \xrightarrow{c} s \right) \bigcup_{v_y \in V \,|\, y \in \text{vars}(e)}^{v_x \in V} T_{\hat{x} \wedge e}^{s,t} \right)$$

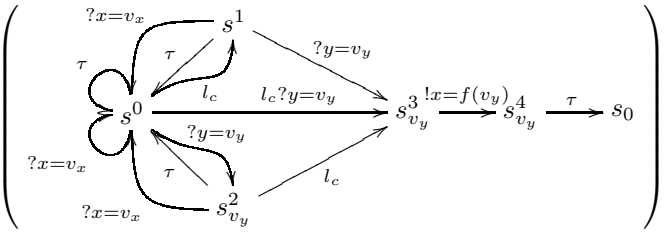
Clock expressions must be rewritten to fit the definition of T_e:

$$\hat{x} = e \wedge f \stackrel{\text{def}}{=} (\hat{x} = \hat{y} \wedge \hat{z} \,|\, \hat{y} = e \,|\, \hat{z} = f) \,/yz$$
$$\hat{x} = e \vee f \stackrel{\text{def}}{=} (\hat{x} = \hat{y} \vee \hat{z} \,|\, \hat{y} = e \,|\, \hat{z} = f) \,/yz$$
$$\hat{x} = e \setminus f \stackrel{\text{def}}{=} (\hat{x} = y \,|\, \hat{y} = e \vee f \,|\, \neg y = f) \,/y$$

Equations A partial equation $c \Rightarrow x = f(y)$ synchronizes x to the value of f by y at the clock c. But x may also be present when either c or y is absent. Therefore, the automaton requires x to be emitted with the value $f(v_y)$ only after the events y and c have occurred. If at least one of either c or y is present, then x may or may not be present with some value u computed by another partial equation. The semantics (combinatorially) generalizes to the case of $c \Rightarrow x = f(y_{1..n})$ with $n \geq 0$.

$A_{c \Rightarrow x = f(y)}$

$$\stackrel{\text{def}}{=} \left(s^0, \{s^{0..1}, s^{2..4}_{v_y}, \mid v_y \in V\}, \{x, y\} \cup \text{vars}(d), \tau, \right.$$
$$\left. \bigcup_{v_x, v_y \in V}^{v_z \in V \mid z \in \text{vars}(c)} \right.$$



$$\left. \right)$$

Structuration *Composition $p \,|\, q$ and restriction p/x* are defined by structural induction starting from the previous axioms with

$$A_{p \,|\, q} \stackrel{\text{def}}{=} A_p \,|^c A_q \qquad A_{p/x} \stackrel{\text{def}}{=} (A_p)/x$$

3 GME

GME is a configurable UML-based toolkit that supports the creation of domain-specific modeling and program synthesis environments [1]. It is developed by the ISIS institute at Vanderbilt University, and is freely available at [11]. Metamodels are proposed in the environment to describe modeling paradigms for specific domains. Such a paradigm includes, for a given domain, the necessary basic concepts in order to represent models from a syntactical viewpoint to a semantical one. It also includes all relationships between those concepts, their organization, and all rules governing the construction of models.

Note *To avoid any confusion between* SIGNAL *and* GME *concepts, the following convention is adopted in the rest of this paper: words beginning with a capital letter refer to* GME *concepts, and those in italics refer to concepts of our metamodel. Mainly, be careful with the*

notion of model.

To use GME, a user first needs to describe a modeling paradigm by defining a project using the MetaGME paradigm. This paradigm is distributed with GME. All modeling paradigm concepts must be specified as classes through habitual UML class diagrams. To build these class diagrams, MetaGME offers some predefined UML-stereotypes [13], among which we use only the following in our metamodel: First Class Object (FCO), Model, Set, Atom, Reference, and Connection. FCO constitutes the basic stereotype in the sense that all the other stereotypes inherit from it. It is basically used to represent abstract concepts (represented by classes). Atoms are elementary objects in the sense that they cannot include any sub-part, while the Model is used for classes that may be composed of various FCOs. In a different way, a class with the Set stereotype can contain a sub-set of FCOs registered in the same Model. A Reference is a typed pointer (as in C++), which refers to another FCO. The type of the pointed FCO is indicated on the metamodel by an arrow (in FIG. 1, the *SignalRef* reference points to a *Signal*).

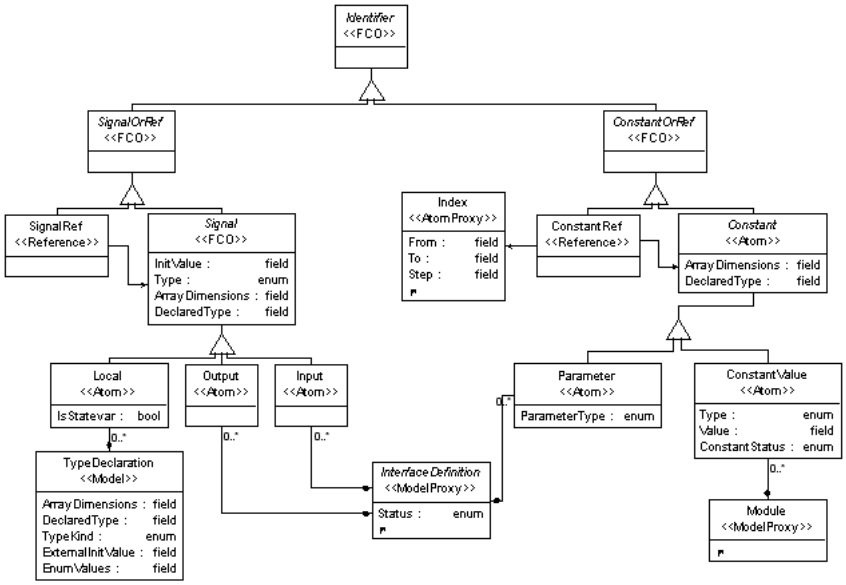


Figure 1: Signal-Meta's Identifier class diagram

There are different kinds of relations that can be expressed between classes, which use these stereotypes. First, the Containment relation is characterized on the class diagram by a link ending with a diamond on the container side. Such a link is shown in FIG. 1 for example between the *Input* atom and the *InterfaceDefinition* Model. Inheritance relations can be represented as in UML or with two other operators: implementation inheritance and

interface inheritance. In implementation inheritance, the subclass inherits all of the base class' attributes, but only those containment associations where the base class functions as the container. At the opposite, interface inheritance allows no attribute inheritance but does allow full association inheritance, with one exception: containment associations where the base class functions as the container are not inherited. All the other types of relationship are specified by classes that use the Connection stereotype.

There are different kinds of relations that can be expressed between classes, which use these stereotypes. First, the Containment relation is characterized on the class diagram by a link ending with a diamond on the container side. Such a link is used in FIG. 1 for example between the *Input* atom and the *InterfaceDefinition* Model. Inheritance relations can be represented as in UML or with two other operators: implementation inheritance and interface inheritance. In implementation inheritance, the subclass inherits all of the base class' attributes, but only those containment associations for which the base class functions is the container. At the opposite, interface inheritance allows no attribute inheritance but does allow full association inheritance, with one exception: containment associations for which the base class functions is the container are not inherited. All the other types of relationship are specified by classes that use the Connection stereotype.

In FIG. 1, some FCOs use a stereotype suffixed by "Proxy", such as Module that uses "ModelProxy". Such stereotypes are references inside the metamodel to a FCO declared in another paradigm sheet. To complete these class diagrams, attributes can be added to classes. These attributes are typed: BooleanAttribute, EnumAttribute that corresponds to a finite list of choices, and FieldAttribute that is a typed text field (string, integer or double).

In these class diagrams, GME provides a means to express the visibility of FCOs within a model through the notion of Aspect (i.e. one can decide which parts of the descriptions are visible depending on their associated aspects). Moreover, it is possible to restrict the use of certain FCOs (add/remove in/from a Model) to a specific Aspect, even if these FCOs are visible in other Aspects.

Finally, OCL Constraints can be added to class diagrams in order to check some dynamic properties on a model designed with this paradigm (e.g. the number of allowed connections associated with a component model). OCL constraints are checked when the events on which constraints are associated with are emitted. There are different kinds of events corresponding to the main action during the modeling, such as create a FCO, connect to a FCO, and change a FCO attribute.

The whole above concepts constitute the basic building blocks that are used to define modeling paradigms in GME. Such a modeling paradigm is always associated with a paradigm file that is produced automatically. GME uses this file to configure its environment for the creation of models using the newly defined paradigm. This is achieved by the **MetaGME Interpreter**, which is a plug-in accessible via the GME Graphical User Interface (GUI). This tool first checks the correctness of the metamodel, then generates the paradigm file, and finally registers it into GME.

Similarly to the MetaGME Interpreter, other components can be developed and plugged into the GME environment. The role of such a component consists of interacting with the graphical designs. To achieve the connection between the component and GME, an executable module is provided with the GME distribution, which enables the generation of the component skeleton. It can be generated in C/C++ or JAVA. In C++, the skeleton is written using the low-level COM language or the **Builder Object Network** (BON) API [13]. GME distinguishes three families of components that can be plugged to its environment: Interpreter, Addon, and PlugIn.

- The role of an Interpreter is to check information, such as the correctness of a model, and/or produce a result, such as a description file. It is the case for the MetaGME Interpreter. An interpreter is applied on user demand and has a punctual execution. Further details are given in Section 6.
- Contrarily to the Interpreter, an Addon is executed as soon as a project is opened, and it works throughout the graphical modeling. An Addon reacts to specific events sent by GME. The GME **Constraint checker** is an example of an Addon. During the description of models, it checks each OCL constraint specified in the used paradigm whenever events to which they relate are emitted by GME.
- Finally, the PlugIn differs from the above two families of components in that it is paradigm-independent. This means that a PlugIn could apply generic operations on models independently of their modeling paradigm. For example, the **Auto-Layout PlugIn** provided with GME has to set the position of each graphical entity to minimize the number of link intersections and to improve the readability of the selected model.

4 SIGNAL metamodel

The SIGNAL metamodel, called Signal-Meta, describes all the syntactic elements defined in SIGNAL v4 [3]. Signal-Meta is composed of several paradigm sheets that define all the relations between the different kinds of signals, SIGNAL operators, and SIGNAL process models. They define as Atom each SIGNAL operator presented in Section 2 and each other one derived from them described in [3]. They also define as Model each SIGNAL container (e.g. process model, sub-process, module), and as Connection each kind of relation between operators and/or identifiers. Section 4.1 gives more details about the representation of Signal-Meta concepts, and Annex A describes all Signal-Meta paradigm sheets. Moreover, to facilitate the modeling, we specify different Aspects presented in Section 4.2. The corresponding division separates mainly the data-flow part and the control part of SIGNAL specifications. To complete this metamodel, OCL constraints defined in Signal-Meta bring some interactivity during the modeling. The list of all current OCL constraints can be found in Annex B. The main goal was to keep as much expressiveness as possible in Signal-Meta than in SIGNAL, and to facilitate user modeling with, for example, n-ary operators.

4.1 Signal-Meta concepts

Among all paradigm sheets, the *Identifiers*' one represented in FIG. 1 defines Atoms for the different kinds of signals (*Input*, *Output*, and *Local*), and constants (*ConstantValue* and *Parameter*). All these Atoms have several attributes including their types, which is an enumeration of all intrinsic types of SIGNAL. The *DeclaredType* attribute is dedicated to a type imported from a SIGNAL library or to a type already declared in GME. The declaration of a new type is done via the *TypeDeclaration* Model. There are different kinds of types, such as enumeration type, structure type or process model type, which are chosen in the *TypeKind* attribute. The way to declare a new type is different according to the kind of the type: for example, a structure type is specified by adding *Local* Atoms in the *TypeDeclaration* Model and by ordering them, whereas all values of an enumerated type have to be specified in the *EnumValues* attribute.

To use in a Model some signal (resp. some constant or index of an iteration) declared in an upper-level Model, one can use a *SignalRef* (resp. a *ConstantRef*) with the same name. Another way for the different Model levels to communicate is to use *Input/Output/Parameter* Atoms. These kinds of Atoms are declared as ports in GME. This means that they are visible in the Model where they are added and in the upper-level Model, so that one can connect them to upper Atoms. *Input/Output/Parameter* Atoms can be added in all Models that inherit from the *InterfaceDefinition* abstract Model.

 Input	 Output	 Local	 Parameter	 ConstantValue
 Delay	 Extraction	 Merging	 Add	 ClockSynchronized

Table 1: Some of Signal-Meta concepts and their icons

The second line of TABLE 1 shows the appearances during the modeling of some FCO representing SIGNAL operators. These appearances are images given in an FCO attribute in the metamodel. *Delay* corresponds to the delay operator, *Extraction* to the sampling operator, *Merging* to the merge operator, *Add* to the addition operator, and *ClockSynchronized* to the clock synchronization operator (all arithmetic, comparison and clock relation operators are represented as for the *Add* Atom with their corresponding symbol). To facilitate the modeling, we add an Atom for boolean expressions and another one for arithmetic expressions in which respectively the boolean expression and the arithmetic expression can be expressed as a textual formula in an attribute.

The main Models of Signal-Meta are *ModelDeclaration*, *SubProcess*, and *Module*. A *ModelDeclaration* corresponds to a SIGNAL process model, which can be either an action, a function, a node, or a process. This choice is done via a *ModelDeclaration* attribute. A *ModelDeclaration* consists of a container in which are declared *Input/Output/Local* signals, static *Parameters*, *ModelDeclaration* and *TypeDeclaration* Models, and in which one can

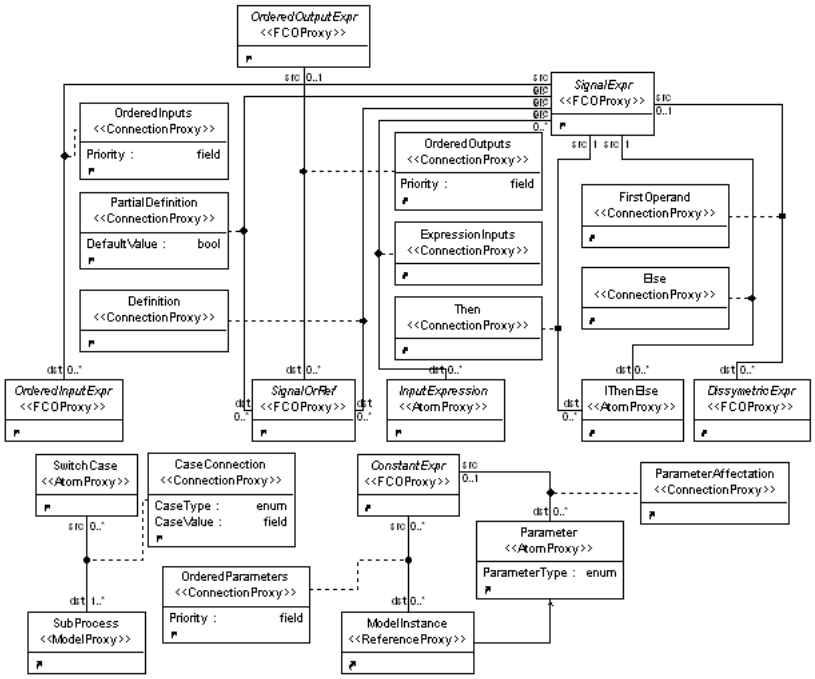


Figure 2: Signal-Meta's 'Expression Connection' paradigm sheet

add FCOs corresponding to SIGNAL operators to express relations between signals. Finally, the *Module* Model is a library of *ModelDeclaration*, *TypeDeclaration*, and *ConstantValue* FCOs. Another interesting point is the way to represent SIGNAL process model instantiations. GME provides a means to express instance objects, thus it would be possible to create instances of *ModelDeclaration* Models. A GME instance of a Model is a deep copy of this Model in which no FCO can be added or removed, but in which attribute values can be modified. To guarantee the exact correspondence between the instance and the corresponding *ModelDeclaration*, we add a Reference, called *ModelInstance*, in Signal-Meta. Thus, *ModelDeclaration* objects are referenced without creating a deep copy of the Model and in a way that guarantees the exact correspondence between the instance and the declaration.

Concerning relations, FIG. 2 corresponds to one of these paradigm sheets and represents all relations between Signal-Meta concepts, except clock relations, which are described in another paradigm sheet. Among them, we can highlight *Definition* whose destination is a *Signal* or a *SignalRef* (gathered in the *SignalOrRef* abstract concept - see FIG. 1), and which allows to specify the definition of a signal. For a given signal, such a Connection can be used only once. SIGNAL offers a means, called partial definition, to avoid the syntactic single assignment rule for the definition of a signal, even if semantically, this rule applies. Similarly,

Signal-Meta offers the *PartialDefinition* Connection to be able to define, in different Models, the different parts of the signal definition.

To simplify the modeling, we make several SIGNAL operators become n-ary operators in Signal-Meta. This is done in different ways according to the operator. For operators of type *OrderedInputExpr* (e.g. *Merging*), we use *OrderedInputs* Connections that have a *Priority* attribute whose value allows to order the incoming Connections. Operators of type *InputExpression* (e.g. arithmetic) are divided into two categories: associative and commutative operators, and the other ones (*DissymetricExpr*) for which the first element needs to be identified (e.g. the substraction operator). The first category uses only *ExpressionInputs* Connections, while the second one uses *FirstOperand* to identify the first element.

4.2 Aspects

Signal-Meta organizes its concepts in four Aspects: *Interface*, *Dataflow*, *ClockAndDependence*, and *Library*. The *Interface* Aspect is dedicated to represent input/output signals of a *ModelDeclaration* and its static parameters. Moreover, a *Specifications* Model can be added to describe clock and dependence relations between these signals. Signals and parameters are ordered according to their position in this Aspect. The *Dataflow* Aspect is dedicated to design all computations of the process and its data flow, whereas the *ClockAndDependence* Aspect contains all clock and dependence relations between signals, instantiations, and subprocesses. Thus, the latter contains mainly clock constraint and relation operators (e.g. *ClockSynchronized*, *ClockUnion*), the *Dependence* Atom, *SignalOrRef*s, and all Connections to link them. The *Dataflow* Aspect can contain all other SIGNAL operators.

This separation of concerns, also recommended by Jackson in [12], makes the modeling more readable. Indeed, Connections in the *Dataflow* Aspect represent data flows, while they represent only relations in the *Clock Relation*. However, this separation between the data-flow and the control parts is not so obvious in SIGNAL. Actually, SIGNAL primitives implicitly express clock relations between their input/output signals, for example the delay and arithmetic operators synchronize automatically their inputs and their outputs. Thus, operators in the *Dataflow* Aspect also express the control part of the process.

Finally, a *Library* Aspect is dedicated for the concept of *Module*. Indeed a *Module* does not express the data-flow or the control of a process. It only corresponds to a library of constant, type, and process model declarations.

4.3 OCL constraints and extension

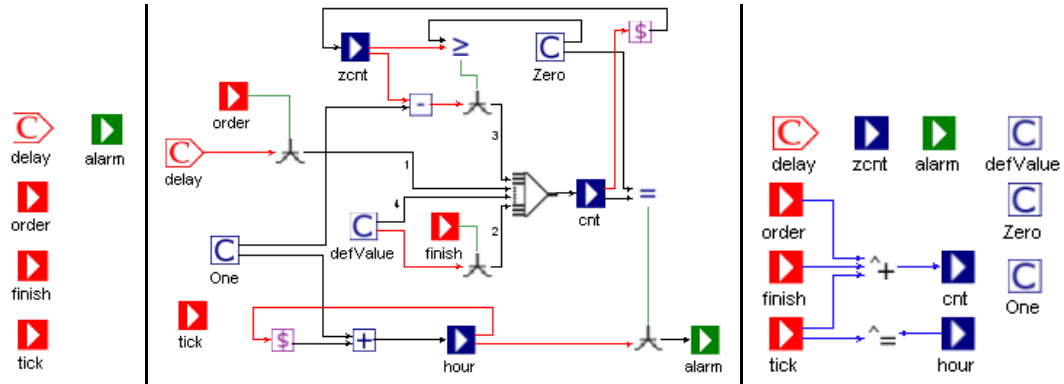
To be able to give some specific information during the modeling, we specify a number of OCL constraints to Signal-Meta. They are mainly used to check the coherence of the values of FCO attributes. For example, one constraint checks that, if the value of the *NumberOfInstants* attribute of the *Delay* Atom is a number and not a name of a signal, this number is not negative. Another constraint checks that the values of the *Priority* attribute for all *OrderedInputs* Connections with the same destination FCO are different. To

complete these constraints, other constraints are automatically generated by the MetaGME Interpreter to check the cardinality affected to each relation.

To facilitate the modeling, we have defined all processes considered intrinsic by SIGNAL in a GME library. In fact, this library contains three *Modules* and one *ModelDeclaration* to represent all usual mathematical functions (e.g. cosine, sine), specific functions to manage complex number and to read (resp. write to) the standard input (resp. output). To represent these functions, we have only defined their interface, which is enough to create *ModelInstance* FCO that refers to them, and to connect their input/output signals and parameters.

5 Example

Here, we apply the metamodel presented in the previous Section to the design of a classical watchdog example. The goal of this watchdog process is to control that some action process is executed within some `delay`. At each time, the action process emits an `order` signal when it begins its execution, and a `finish` event when it finishes it. If the job is not finished in time, the watchdog must emit an `alarm` signal to indicate at what time an error occurs. Moreover, if a new `order` occurs when the previous one is not finished, the time counting restarts from zero. A `finish` signal out of delay, or not related to an `order`, will be ignored.



(a) *Interface* (b) *Dataflow* (c) *ClockAndDependence*

Figure 3: The Watchdog example

The `Watchdog` process can be specified in GME as shown in FIG. 3. FIG. 3(a) represents the Interface Aspect in which are described the input/output signals and the static parameters of the process. Thus, one has to drag and drop an Input Atom (red boxes) for the `order` and `finish` signals, and an Output Atom (green boxes) for the `alarm`. In order to count the time, another input signal called `tick`, which must be provided at regular interval, is added to the Interface Aspect to represent each tick of a clock. Finally, the `delay` to process an

order is expressed as a number of `tick`s by a Parameter Atom. The types of each of these signals/parameters are specified in the attributes of the corresponding Atom.

In the Dataflow Aspect (FIG. 3(b)), three local signals are declared: `hour`, `cnt`, and `zcnt`. The `hour` signal represents the internal clock to count the time. The `cnt` signal works as a countdown before emitting an alarm: when `cnt` is 0, the alarm is emitted with the value of `hour`. The value of `cnt` is fixed, by order of priority, to: (i) `delay` when an order is emitted, (ii) `defValue` when `finish` is emitted, (iii) the previous value of `cnt` contained by `zcnt` decremented by one, or finally to (iv) `defValue`. This order is fixed using the *Priority* attribute of all incoming Connections on the *Merging* Atom. In FIG. 3(b), red links connect their source FCO as first operand of the destination FCO, green links connect a boolean expression to an *Extraction* Atom, black links whose destination is a *Signal* Atom correspond to a *Definition* of this signal, and finally other black links are Connection specific for each operators (cf. FIG. 2).

In the Clock Aspect (FIG. 3(c)), `hour` and `tick` are synchronized using the *ClockSynchronized* Atom. This leads `hour` to be incremented at each `tick`. Moreover, `cnt` has to be present each time one of the input signals is present. This is expressed with the *ClockUnion* Atom whose result is affected to `cnt`.

6 Model Interpretation

Taking advantage of the ease of modeling in GME, we make GME become a front-end for POLYCHRONY, the current development platform for SIGNAL, through the use of Signal-Meta. Then, we need to transform the graphical specifications using Signal-Meta to the corresponding SIGNAL programs. Therefore, we have implemented a GME Interpreter for Signal-Meta models, which acts similarly as the MetaGME Interpreter for MetaGME metamodels. Figure 4 represents the different steps during the interpretation. There are three main steps:

Step 1: Tree generation. Each FCO selected in the GME GUI is associated with a tree (the intermediate representation in FIG. 4) whose root is the selected FCO. Each node of these trees corresponds to a SIGNAL process model, and each leaf to a symbol (e.g. signal, constant) in the generated program. The tree is built by recursive instantiations of each node into BON objects [13] according to their type in the metamodel. The root FCO is first instantiated. Then, all its contained Models and FCOs, which correspond to symbols (e.g. *Input*, *Output*), are instantiated. The same process is applied recursively on each sub-Model. For example, the instantiation of a *Module* Model results in the instantiation of its contained elements among which *ModelDeclaration*, *TypeDeclaration*, and *ConstantValue* FCOs. While *ConstantValue*s are only Atoms, and thus leafs of the tree, *ModelDeclaration*s and *TypeDeclaration*s contain subparts. In the same manner, each of these container elements recursively instantiates its own symbols and SIGNAL process models.

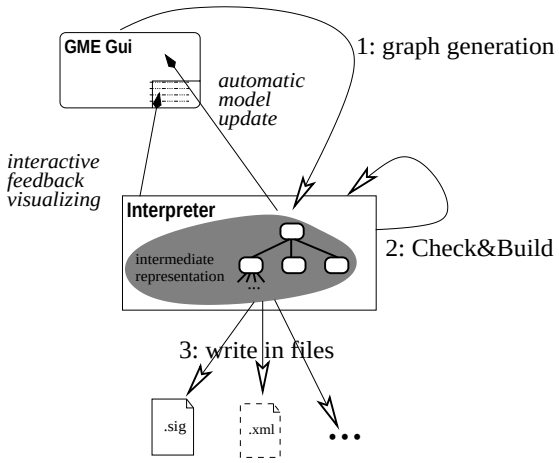


Figure 4: From GME to SIGNAL files.

Step 2: Check&build. This step consists in building the SIGNAL equations for each node of the tree created at the previous step. Each Model (*ModelDeclaration*, *SubProcess*, etc.) has to build the equations corresponding to each element it contains.

To produce these equations, we need to analyze all concepts and all relations (i.e. Connections) between them. This analysis consists in visiting each node of a directed graph whose arcs are Connections. To analyze the graph, we need to select start/end points, which allows, as much as possible, to avoid visiting the same path twice. We call **end-statements** these start/end points. Basically, it corresponds to all named elements (e.g. signal, declaration of model, model instance). Actually, the analysis consists first in producing the SIGNAL code corresponding to the end-statement from which the analysis starts; then if there are specific Connections (for example *Definition*) whose source is the starting FCO, the analysis follows them in the backward direction and tries recursively to produce the SIGNAL code corresponding to the FCO(s) which is(are) the source of this(these) connection(s). The analysis is stopped when the source of a Connection is an end-statement or when an error, such as cycle or FCO without a needed Connection, is detected. More precisely, inside a Model, an equation is produced:

- for each Connection of type *Definition*, *PartialDefinition*, and *ConstraintInputs* whose destination is either a *Local* or an *Output* Atom (or *SignalRef* Reference which points to such an Atom). As shown in FIG. 3(b), taking the example of the `hour` local signal, there is a *Definition* Connection whose destination is `hour` and whose source is the *Add* Atom. Then, the analysis follows the two Connections whose destination is the *Add* Atom. The first one leads to the `One` *ConstantValue*, which is an end-statement, thus the analysis stops. The second one leads to a *Delay* Atom, thus the analysis needs to continue following the Connection linked to the *Delay* Atom. This leads finally to

an end-statement, which is the `hour` local signal itself. So the produced equation is `hour := (One + (hour$(1) init (0)))`.

- for each Atom representing a clock constraint or a dependence relation. For example, in FIG. 3(c), the *ClockSynchronized* Atom is the destination of two *ConstraintInputs* Connections: one from the `tick` signal and one from the `hour` signal. As result, the equation `tick ^= hour` is produced.
- for each *ModelInstance* Reference, which refers to a *ModelDeclaration* that indicates the model to instantiate. To produce an equation of a SIGNAL process instantiation, an intermediate signal is generated for each *Input/Output/Parameter* FCO of the referred Model. For each Connection to these FCOs, an equation is created using the intermediate signal as the Connection destinations.
- for each *TypeDeclaration* Model. According to the kind of type declaration, the analysis is different: for enumeration types, we use the *EnumValues* attribute in which there is one value per line; for structure types, all *Local* Atoms are listed; for model types, all *Input/Ouput/Parameter* Atoms are listed and the corresponding interface is generated; finally, for external types, the content of the *DeclaredType* attribute is used.

Input/output signals and parameters are ordered in the interface of a SIGNAL model according to the position of their corresponding Atoms in the *Interface* Aspect.

In the same step before the equation generation, some corrections could be applied to the graphical Model, for example, when a Reference points to an FCO that is not declared in the same scope as the Reference. In this situation, the properties of the corresponding graphical components are systematically updated.

As soon as an error is encountered during this second step, a message is displayed in the GME console, indicating FCOs concerned by the error as HTML links. Whenever the user clicks on a link, the corresponding graphical object is automatically displayed. This is very convenient to make rapid corrections.

Step 3: Dump in files. The third and last step consists in visiting one more time each node of the tree and writing the corresponding equations into destination files at the relevant place in the SIGNAL model. The declarations of signals, constants, and labels are built and added at the same time. The code of FIG. 5 corresponds to the application of our interpreter on the watchdog example described in FIG. 3.

As a global remark, we have to mention that the interpretation process can only be applied to higher-level Models. We impose this restriction in order to be sure that the selected Models do not use signals declared at an upper level in the hierarchy of a Model. So, the interpreter only generates a file for selected Models, which are immediate children of the Root Folder (i.e. the root of the current project).

Finally, we can notice that the second and the third steps can be specialized. The interpreter generates files using the SIGNAL syntax. However, it is possible to specialize the interpreter to construct equations using, for example, XML syntax.

```
process Watchdog =
   { integer delay; }
   ( ? integer order;
       event finish;
       event tick;
     ! integer alarm; )
   (| alarm := (hour when (cnt = Zero))
    | hour := (One + (hour$(1) init (0) ))
    | cnt ^= (order ^+ tick ^+ finish)
    | cnt := ((delay when ^order)
             default (defValue when finish)
             default ((zcnt - One) when (zcnt >= Zero))
             default defValue)
    | zcnt := (cnt$(1) init (-1) )
    | tick ^= hour
    |)
   where
      constant integer One = (1);
      constant integer defValue = (-1);
      constant integer Zero = (0);
      integer hour;
      integer cnt;
      integer zcnt;
   end; % process Watchdog
```

Figure 5: Code generated by the Interpreter on the watchdog example

7 Discussion

The modeling paradigm introduced in this paper constitutes the first work for generalizing the use of formal methods proposed by POLYCHRONY. This approach is developed using metamodels to achieve a relative independence from the modeling platform. The higher their abstraction expression level is, the more adaptable to various operational environments they will be. Indeed, Model Driven Software Development is based on a number of common standards such as XMI, OCL and UML, that can be mapped onto different environments. Thus, we have chosen GME to develop the metamodel, because it is indeed a good solution to create a metamodel quickly, and it offers automatically a customized modeling environment.

The metamodel combined with the Interpreter makes from Signal-Meta a new front-end for POLYCHRONY. Actually, Signal-Meta and its interpreter check only structural information of the graphical specification, such as cyclic definitions or the well-formedness of the modeling. There is no verification of types and clock constraints. The SIGNAL compiler is in charge of these verifications. One of our future goals is to obtain a graphical and fully interactive modeling under GME. Currently, the interaction is limited to structural errors

detected by the OCL constraints added to Signal-Meta. We should extend the modeling so that the environment is able to check deeper semantic errors and to display them graphically and dynamically during the modeling.

Anyway, to really generalize the use of formal methods, our metamodel must be accessible in more popular frameworks, such as Eclipse. The ATLAS Group from INRIA [2] has realized a bridge between GME and the Eclipse Modeling Framework (EMF) [8]. Some transformations [4] have been developed between MetaGME metamodels and EMF metamodels. However, these transformations keep only concepts and relations between them, they do not cover all features offered by GME: for example, all informations concerning Aspects disappear. With this restriction, all metamodels realized with GME, and particularly for our interest Signal-Meta, can be transformed to metamodels under EMF. However, it is important to note that the current Interpreter uses the BON API and so is dedicated to GME. Thus, it must be specifically developed for Eclipse.

This metamodel can be considered as a first effort toward the development of a more general-purpose UML profile for modeling real-time and embedded systems, called MARTE [15]. Moreover, Signal-Meta constitutes a kernel to create environments for multi-clock systems. Signal-Meta has already been extended for different purpose. A first extension has been done to model multi-clock mode automata [6]. In this work, the automaton describes the control of the systems, and in each state of the automaton, SIGNAL equations are built in the Signal-Meta way. The interpreter was also extended to transform such an automaton into the corresponding SIGNAL program.

Another extension, called MIMAD [5], concerns the design of avionics systems based on the Integrated Modular Avionics (IMA) architecture develop around the APEX-ARINC 653 standard. Initially, some predefined services have been implemented in a POLYCHRONY library [10]. All these services and all levels of the IMA Architecture extend Signal-Meta to build MIMAD. Each level of the IMA Architecture is represented and inherits from Signal-Meta FCO, which allows to reuse easily features of the interpreter. Signal-Meta are mainly used in MIMAD to represent specific user-designed functions and the data flow between the input/output signals of IMA levels. The interpreter has also been extended to produce the SIGNAL program using processes developed in POLYCHRONY.

8 Conclusions

In this paper, we have presented Signal-Meta, the metamodel of the SIGNAL language developed in GME and its Interpreter to transform the graphical specifications into SIGNAL programs. Both tools make from GME a new front-end for the POLYCHRONY workbench. Moreover, Signal-Meta has already been used as foundations to build more specialized multi-clocked environment such as for mode automata and for avionics system design.

As discussed in Section 7, Signal-Meta and its interpreter check only structural information. There is no type checking or clock constraint verification. To complete the modeling under GME and to overcome these limitations, one of the possible way is to interface directly POLYCHRONY as a GME Addon. Thus, the internal representation of the SIGNAL compiler

could be produced automatically during the modeling, and then give access to possible clock or type problems.

References

[1] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 8–15, New York, NY, USA, 2003.

[2] ATLAS Group (INRIA & Lina, Université de Nantes). ATL, ATLAS Transformation Language, Reference site. `http://www.sciences.univ-nantes.fr/lina/atl/`.

[3] Loic Besnard, Thierry Gautier, and Paul Le Guernic. SIGNAL V4-INRIA version: Reference Manual. `http://www.irisa.fr/espresso/Polychrony/doc/document/V4_def.pdf`.

[4] Jean Bézivin, Christian Brunette, Régis Chevrel, Frédéric Jouault, and Ivan Kurtev. Bridging the Generic Modeling Environment and the Eclipse Modeling Framework. In *Proceedings of the 4th workshop in Best Practices for Model Driven Software Development, OOPSLA*, October 2005.

[5] Christian Brunette, Romain Delamare, Abdoulaye Gamatié, Thierry Gautier, and Jean-Pierre Talpin. A Modeling Paradigm for Integrated Modular Avionic Design. Technical Report RR-5715, INRIA, October 2005.

[6] Christian Brunette and Jean-Pierre Talpin. Compositional modeling and transformation of multi-clocked mode automata. Technical Report RR-5728, INRIA, October 2005.

[7] Consortium, Sacres. The Declarative Code DC+, Version 1.4. `ftp://ftp.irisa.fr/local/signal/publis/research_reports/dc+.ps.gz`, november 1997.

[8] Eclipse Modeling Framework. Reference site. `http://www.eclipse.org/emf/`.

[9] ESPRESSO-IRISA. POLYCHRONY website. `http://www.irisa.fr/espresso/Polychrony`.

[10] Abdoulaye Gamatié and Thierry Gautier. Synchronous Modeling of Modular Avionics Architectures using the SIGNAL Language. Technical Report RR-4678, INRIA, 2002.

[11] Institute for Software Integrated Systems (ISIS). Vanderbilt University. The Generic Modeling Environment (GME). `http://www.isis.vanderbilt.edu/Projects/gme/`.

[12] Ethan K. Jackson and Janos Sztipanovits. Using separation of concerns for embedded systems design. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 25–34, New York, NY, USA, 2005. ACM Press.

[13] Akos Ledeczi, Miklos Maroti, and Peter Volgyesi. The Generic Modeling Environment. In *Proceedings of the IEEE Workshop on Intelligent Signal Processing (WISP'01)*, May 2001.

[14] Hervé Marchand, Patricia Bournai, Michel Le Borgne, and Paul Le Guernic. Synthesis of Discrete-Event Controllers based on the SIGNAL Environment. In *Discrete Event Dynamic System: Theory and Applications*, volume 10 of *4*, pages 325–346, October 2000.

[15] OMG. UML Profile for modeling and analysis of real-time and embedded systems (MARTE). OMG document realtime/05-02-06.

[16] Dumitru Potop-Butucaru and Benoit Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Proceedings of the Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, pages 48–57. IEEE Press, 2005.

[17] Jean-Pierre Talpin, Dumitru Potop-Butucaru, Julien Ouy, and Benoit Caillaud. From multi-clocked synchronous processes to latency-insensitive modules (short paper). In *Proceedings of the fifth ACM International Conference on Embedded Software (Emsoft)*, pages 282–285, Jersey City, NJ, USA, September 2005. ACM Press.

Annexes

A Signal-Meta paradigm sheets

This Section describes all paradigm sheets of Signal-Meta showing at the same time the class diagram and the declaration of Aspects. The reader is referred to [3] for a precise description of the corresponding syntactic structures.

A.1 Containers

FIG. 6 specifies all abstract and non-abstract Models of Signal-Meta. The main kinds of Models, characterized by abstract Models, are Models with an *Interface* Aspect (*InterfaceDefinition*), with a *ClockAndDependence* Aspect (*ModelsWithClockRelations*), and/or with a *Dataflow* Aspect (*ModelsWithDataflow*). Several Models derive from these classes. For example, *ModelDeclaration* gathers the three kinds. All SIGNAL process models are gathered in the *ModelDeclaration* Model. Its *ModelType* attribute allows to choose between an action, a function, a node, and a process.

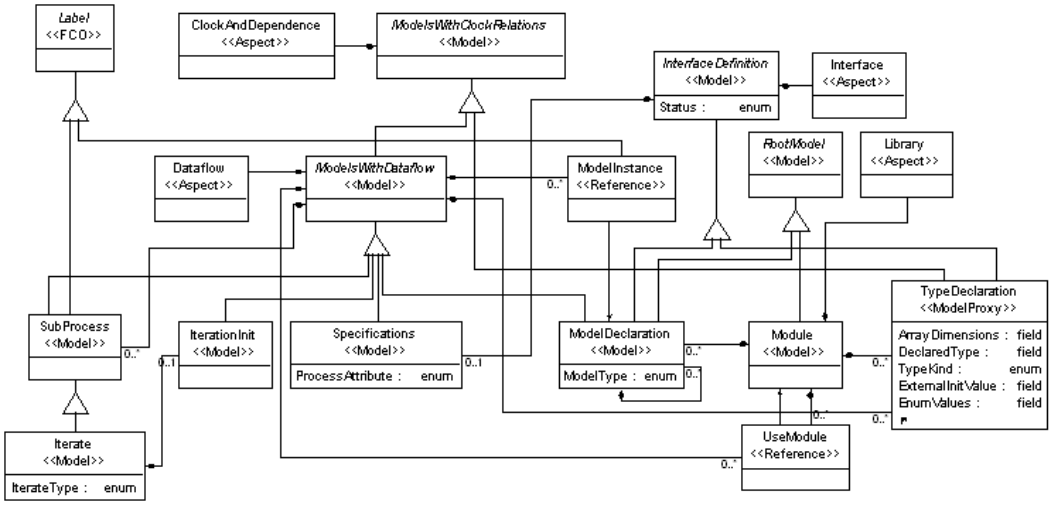


Figure 6: 'Models' paradigm sheet

SubProcess, *Specifications*, and *IterationInit* gather only the *ClockAndDependence* and the *Dataflow* Aspect. *SubProcess* is simply a *ModelDeclaration* without any interface. *Specifications* is used to specify properties between the *Input/Output* signals of the Model that contains it. The *Specifications* Model can be contained by all Models with an *Interface* Aspect. The *Specifications* Model has an attribute (*ProcessAttribute*) to qualify the corre-

sponding model as safe, deterministic automaton, or unsafe.

IterationInit can only be contained by an *Iterate* Model and is used to describe the specifications needed to initialize the iteration. SIGNAL offers two constructions to express iterations: array and iterate. These constructions are gathered in the same *Iterate* Model. The choice between these constructions is made through the *IterateType* attribute.

TypeDeclaration is used to declare any kind of types (see Section 4.1). It inherits from *InterfaceDefinition* to allow the declaration of the interface of model type (action, function, node, and process). It also inherits from *ModelsWithClockRelations* to declare the signals contained by a structure type and to specify the signals and their clock relations for a bundle type (a bundle is a structure whose fields are not synchronized and in which some clock constraints can be specified). Enumerated types are declared by specifying each value of the enumeration in the *EnumValues* attribute. The *ArrayDimensions* attribute serves to declare the dimensions for an array type (e.g. [10]integer), *DeclaredType* to specify the name of any existing type, and *ExternalInitValue* to specify the initial value for any signal that uses an external type. Note also that all Models that inherit from *InterfaceDefinition* have a *Status* attribute to specify the visibility (private or public) inside a *Module*.

A *Module* corresponds to a library of model processes, type declarations, and constants. The *Library* Aspect is dedicated to this Model. And, like *ModelDeclaration*, *Module* inherits from the *RootModel* abstract Model. This means that they can be added to the root of a GME project based on Signal-Meta.

FIG. 6 also specifies two References: *ModelInstance* and *UseModule*. A *ModelInstance* corresponds to an instance of a *ModelDeclaration* (see Section 4.1). It is only through *ModelInstance* that one can define the *Input* signals and the static *Parameter*s. In fact, *Input*s and *Parameter*s can not be defined inside their Model, and a *ModelDeclaration* corresponds only to the declaration of a model process. At the opposite, *Output*s can only be defined inside their Model.

UseModule is used to import external *Module*s. The import of a *Module* is required as soon as a *ModelDeclaration*, a *ConstantValue*, and/or a type declared in an external *Module* is used (except for intrinsic processes).

A.2 Signals and constants

FIG. 7 describes all kinds of *Signal*s and *Constant*s of Signal-Meta or References on them. There are three kinds of *Signal*s: *Input* and *Output* that can only be added to Models with an *Interface* Aspect, and *Local*, which can be added to any *ModelsWithDataflow* or *TypeDeclaration* Models (see FIG. 17). A *Signal* is characterized by a name, and by a type. An OCL constraint guarantees that the name of the signal is unique inside a Signal-Meta Model. The type is either one of those listed in the *Type* attribute (this list corresponds to

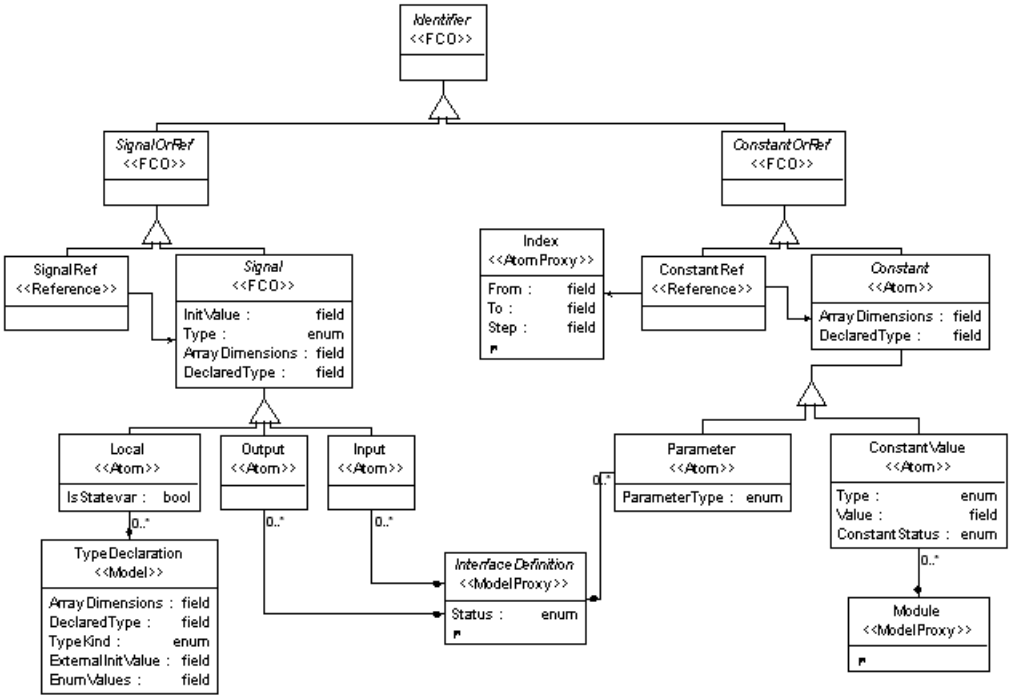


Figure 7: 'Identifiers' paradigm sheet

that given in [3]), or that specified in the *DeclaredType* attribute. As for *TypeDeclaration*, the type can be an array type, thus the dimensions of the array must be specified in the *ArrayDimensions* attribute. This attribute can contain either integer values, or the names of constant values. Finally, *Local* signal has a specific attribute (*IsStatevar*), which indicates if the signal is a state variable.

Similarly, there are two kinds of *Constant*s: *Parameter* that can only be added to Model with an *Interface* Aspect, and *ConstantValue*, which can be added to any *ModelsWithDataflow* Models (see FIG. 17). Like a *Signal*, a *Constant* is characterized by a name that must also be unique inside a Model, and by a type. However, a *Parameter* can be typed by a model type, then its list of types (the *ParameterType* attribute) is extended with the four kinds of model types (action, function, node, and process). And for a *ConstantValue*, a specific *Value* attribute indicates its static value. Finally, we can also note that a *ConstantRef* can refer either a *Constant*, or an *Index*. An *Index* is used by an *Iterate* Model. It is characterized by a starting value (*From* attribute), a final value (*To* attribute), and the value (*Step* attribute) to add at each iteration to go from the starting value to the final one.

A.3 Operators

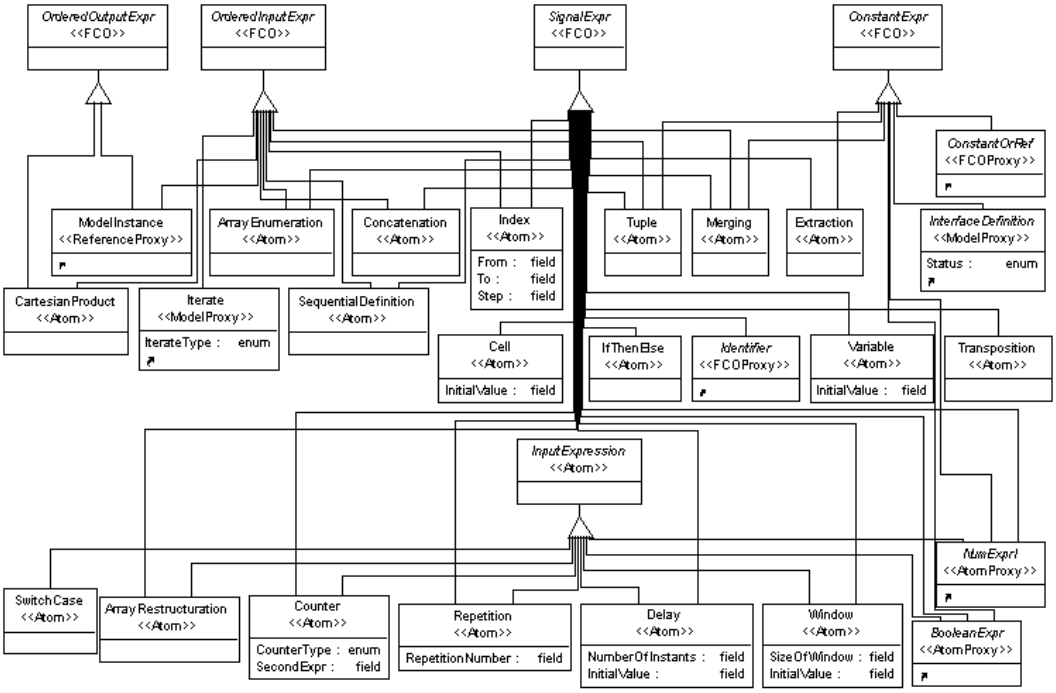


Figure 8: 'Signal expressions' paradigm sheet

FIG. 8 and 9 express a division of SIGNAL operators (except clock relation ones) in different categories according to the Connection that can be linked to them (see FIG. 10). *OrderedInputExpr* (*OrderedOutputExpr*) gathers operators, which require to define exactly the order of each incoming (resp. outgoing) Connection, for example for the *Merging* operator. This order is obtained according to the value of the *Priority* attribute on *OrderedInputs* (resp. *OrderedOutputs*). Thus, all incoming (resp. outgoing) Connections that have the same destination FCO must have different values in their *Priority* attribute.

ConstantExpr gathers all FCOS that can be used to build a constant expression to define for example the value of a *Parameter*. *ConstantExpr* includes mainly constants, arithmetic and boolean operators, and the sampling and merging operators.

InputExpression operators (e.g. arithmetic) are divided into two categories: associative and commutative operators, and the other ones called *DissymetricExpr* (see FIG. 9) for which the first element needs to be identified (e.g. the substraction operator). The first

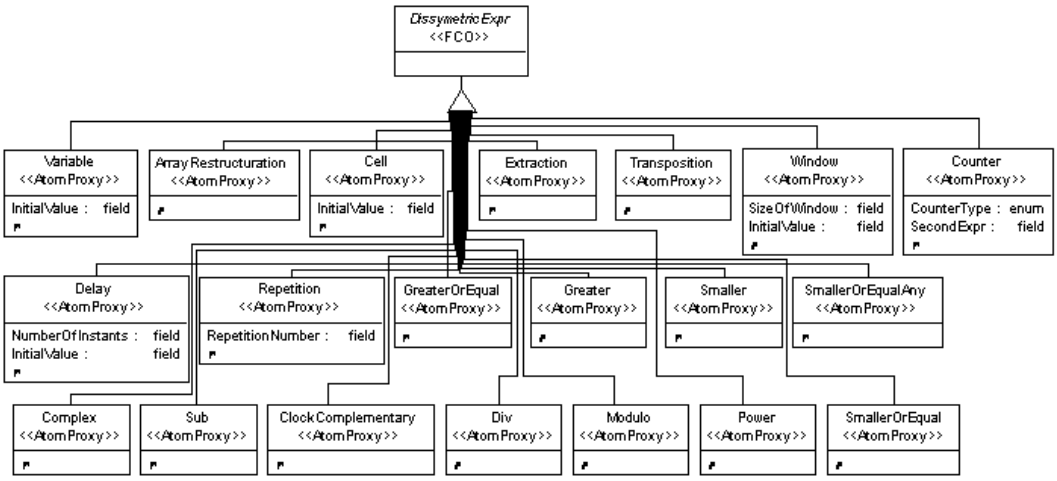


Figure 9: 'Dissymetric Expressions' paradigm sheet

category contains all *InputExpression*operators that do not inherit from *DissymetricExpr*. The first category uses *ExpressionInputs* Connections, while the second one differentiates the first element with a *FirstOperand* Connection.

There are two ways to define a signal. One can use either one single *Definition* or several *PartialDefinition* Connections. For *PartialDefinition*, it is possible to specify a default value by setting the value of the *DefaultValue* to true.

There are two choice operators: *IfThenElse* and *SwitchCase*. The *IfThenElse* operator gives a choice between two alternative expressions for the definition of a signal. This Atom has to be the destination of at least one boolean condition, exactly one *Then* Connection and one *Else* Connection. The boolean condition is expressed, as for any *ConditionnedExpr* (Fig. 12), by one (or several) *Condition* Connection linked to a *BooleanExpr* and/or by one (or several) *WhenSignal* Connection linked to a *SignalOrRef*. The *WhenSignal* Connection has been introduced to be able to test the presence of a signal or the value of a boolean signal without creating an intermediate boolean expression. Thus, the *WhenSignal* Connection has an attribute (*SignalState*) that specifies the kind of test: test the presence, test if the boolean signal is $True/False$.

The *SwitchCase* operator represents an expression of processes that allows to compose definitions according to the different values of a signal. The type of the signal must be either integer or an enumerated type. An *ExpressionInputs* Connection links the *SwitchCase* to the corresponding signal. A case is represented by a *SubProcess* that is linked to the *SwitchCase* by a *CaseConnection*. This Connection has two attributes: *CaseType* to choose the kind of conditions (i.e. giving an enumeration of values, an interval, or an else case),

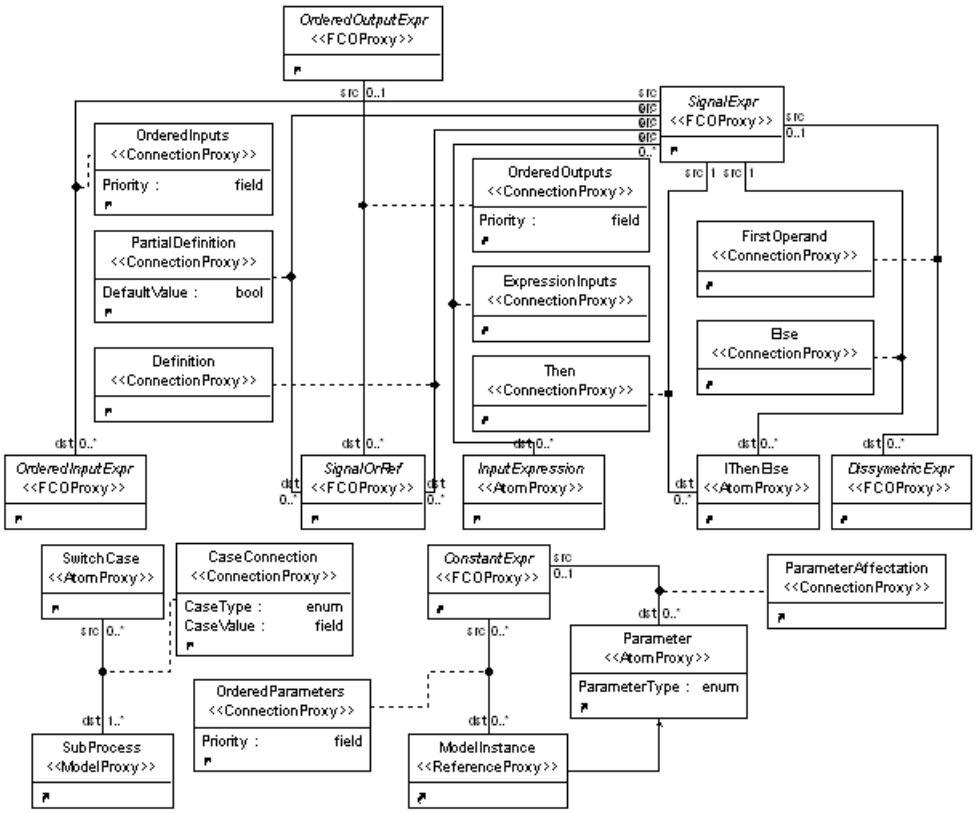


Figure 10: 'Connections between expressions' paradigm sheet

and *CaseValue* to express either the enumeration of values or the interval.

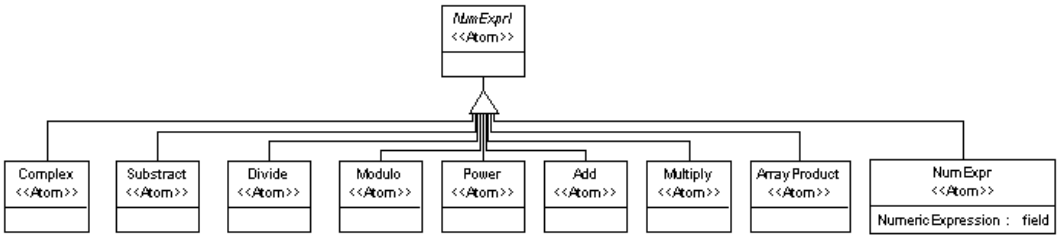


Figure 11: 'Numeric Expressions' paradigm sheet

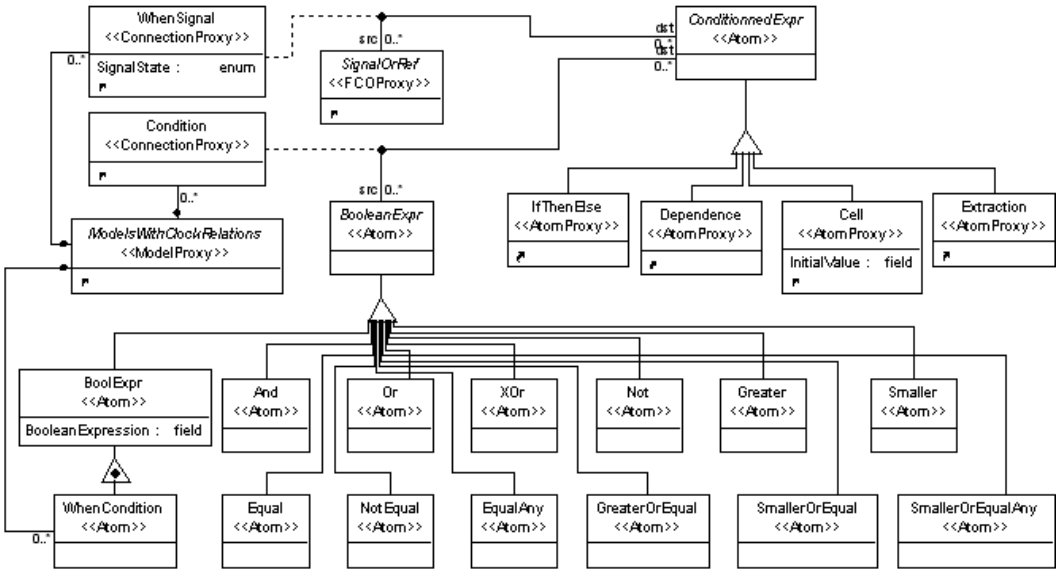


Figure 12: 'Boolean expressions' paradigm sheet

FIG. 11 and 12 describe respectively all arithmetic and boolean operators. For both, there is a specific Atom (*NumExpr* and *BoolExpr*) that has been added to ease the modeling by just writing the arithmetic (resp. boolean) expression in the attribute of that Atom. Moreover, for boolean expression, we give access to such an Atom in the *Dataflow* Aspect (*BoolExpr* Atom) and in the *ClockAndDependence* Aspect (*WhenCondition* Atom).

A.4 Relations

FIG. 13 describes all relations (i.e. Connections) of Signal-Meta and all attributes carried by them. Among these attributes, we can mention *SrcField* (resp. *DstField*), which gives access, for signals of type structure or bundle, to the name of the signal (inside the structure or the bundle type), which is at the source (resp. destination) of the Connection .

CastType specifies the type for the cast of the value of the signal at the source of the Connection.

LastIterationValue and *ArrayRecovery* are attributes that are only used inside an *Iterate* Model. *LastIterationValue* specifies that we are interested by the value of the previous iteration or by the value of the signal for the current instant. *ArrayRecovery* allows to specify the expression returned if the iteration tries to access to an index out of the array dimensions.

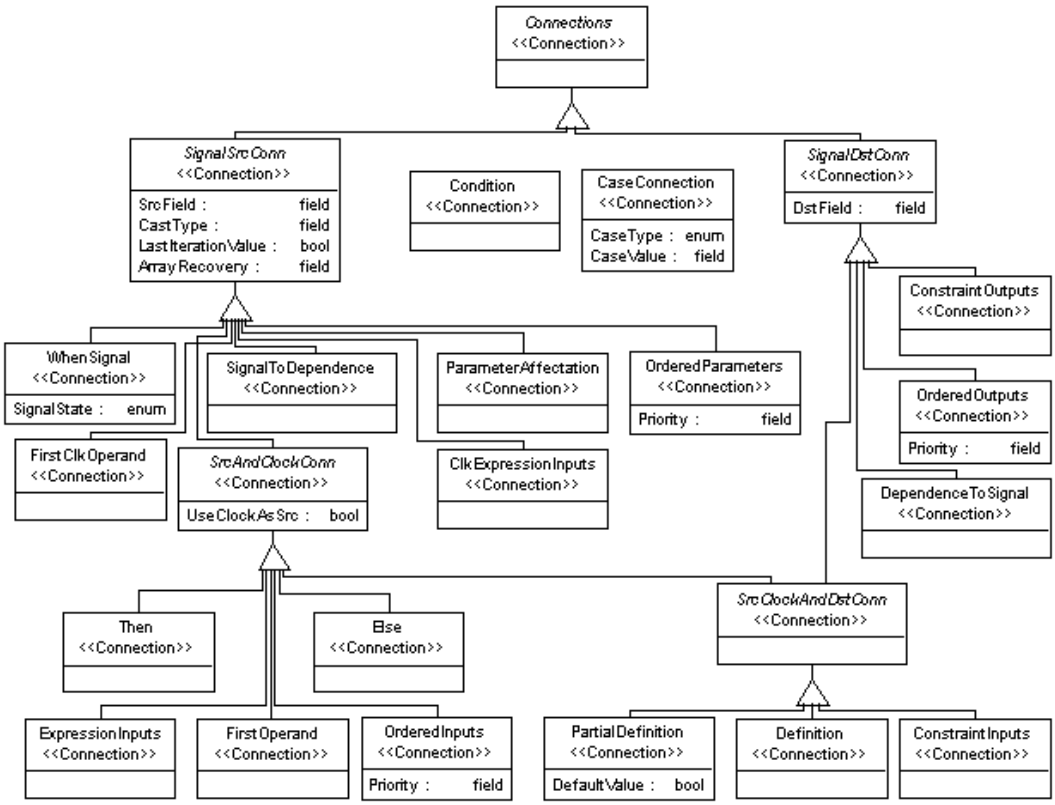


Figure 13: 'Connections' paradigm sheet

Finally, *UseClockAsSrc* is a boolean attribute that indicates: (i) if it is $True$ that the clock of the signal at the source of the Connection is used, (ii) if it is $False$ that the value of the signal is used.

A.5 Clock relation and constraint operators

FIG. 14 describes all clock relation and clock constraint operators and how they can be used. Thus, we can express the union of clocks (*ClockUnion*), their intersection (*ClockIntersection*), and their difference (*ClockComplementary*). It is possible to synchronize two (or more) signals (*ClockSynchronized*), to specify that two (or more) signals are never present at the same instant (*ClockExclusion*). One can also constraint two (or more) signals to be synchronized and to have the same value at each instant (*ClockIdentity*), or specify that the clock of one (or more) signal is faster (resp. slower) than another one using the *ClockGreater*

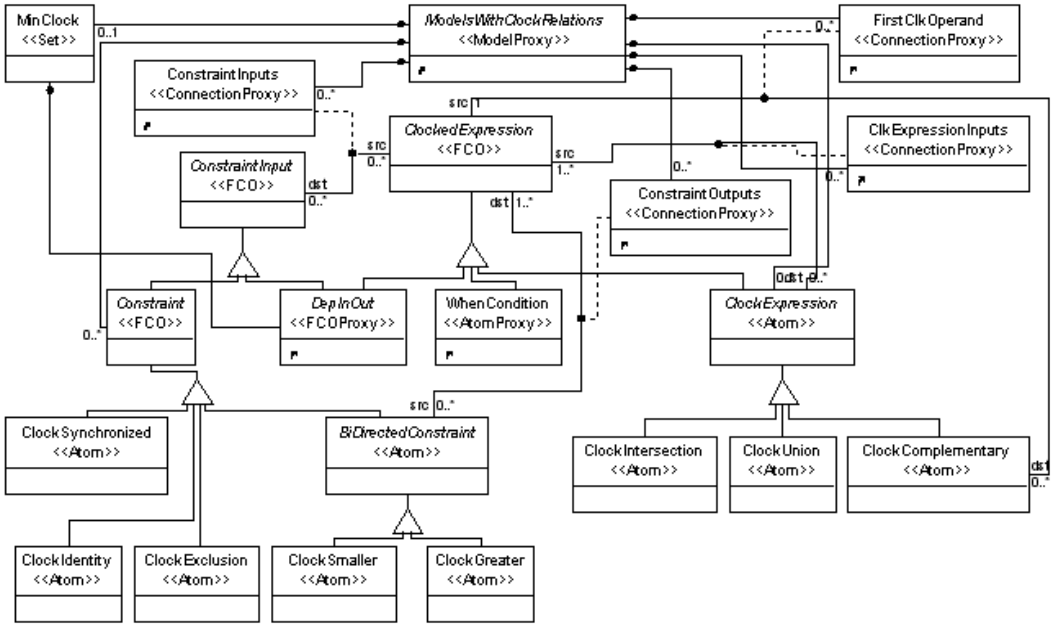


Figure 14: 'Clock Expressions' paradigm sheet

(resp. *ClockSmaller*) Atom.

These clock constraint operators can be applied on all FCOs that inherits from the *DepInOut* abstract class (see Fig. 15). This means that they can be applied to signals (or *SignalRef*), and on each FCO which inherits from *Label*: *ModelInstance* and *SubProcess* (see Fig. 6). This inheritance means that both FCOs have a name that identifies them. Moreover, the *WhenCondition* Atom declared in Fig. 12 can also participate to clock constraints and clock relations.

We have also added a new functionality to Polychrony to be able to deduce the minimal clock of a signal (different from the null clock) according to the clocks of their partial definition. This is done in Signal-Meta via the *MinClock* Set. This Set contains all FCOs that inherit from the *DepInOut* abstract class and whose clocks need to be deduced.

A.6 Dependences

Fig. 15 introduces the *Dependence* operator, which allows to specify a scheduling for different *Signal*s (or *SignalRef*s), *SubProcess*es, and *ModelInstance*s. These scheduling specifica-

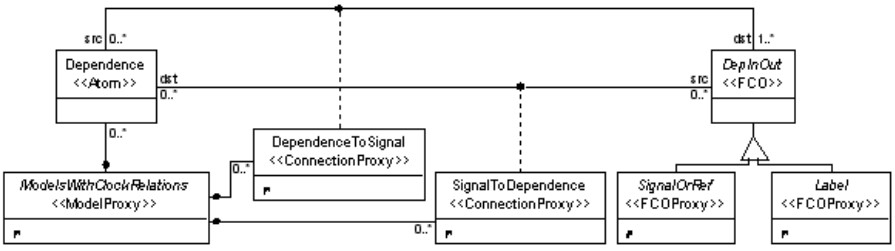


Figure 15: 'Dependence' paradigm sheet

tions can also be conditioned by a boolean expression specified in a *WhenCondition* Atom (see. FIG. 12).

A.7 Assertions and pragmas

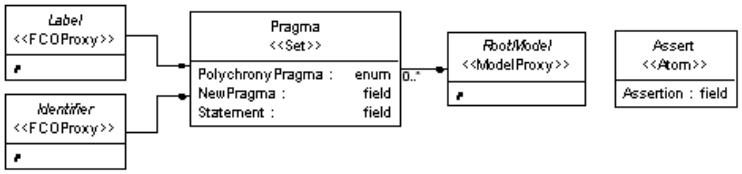


Figure 16: 'Assertion and Pragma' paradigm sheet

FIG. 16 introduces the POLYCHRONY concept of *Pragma*. A pragma has no semantic effect. It can be ignored by a compiler, or it can trigger a specific processing. A pragma is characterized by a name, a list of objects with which it is associated, and a statement.

The name qualifies the kind of *Pragma*. There are different predefined kinds in POLYCHRONY. They are listed in the *PolychronyPragma* attribute. These kinds of *Pragma* allow for example to call external C/C++/Java code, to add some comments in some process models, or to add some external call that are interpreted when translated into the SIGALI representation. Other kinds of pragma can be added by specifying their name in the *NewPragma* attribute. The list of objects with which it is associated is built by listing all FCOs contained in the *Pragma* Set. By default, when the Set is empty, the pragma is associated with the current process model.

FIG. 16 also introduces the intrinsic process of assertion: *Assert*. To use it, one has only to write the assertion expression in the *Assertion* attribute.

Finally, FIG. 17 describes only the FCOs that can be contained by the *ModelsWithDataflow* abstract Model.

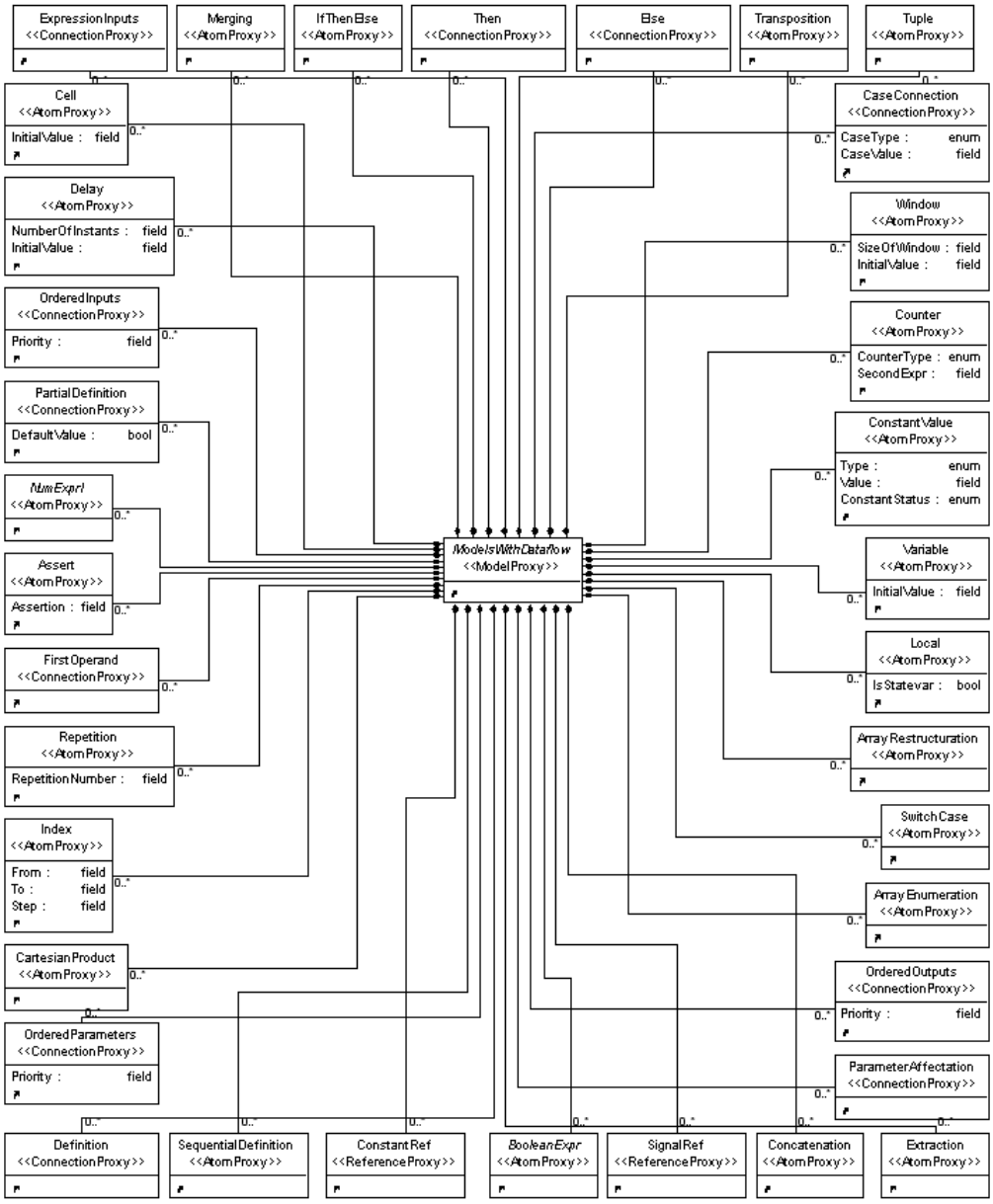


Figure 17: 'ModelsWithDataflow containment' paradigm sheet

B Signal-Meta OCL constraints

In this Section, we describe the OCL constraints added to Signal-Meta and checked during the definition of any Signal-Meta-based application model by the *Constraint Checker*

provided with GME. In GME, an OCL constraint is characterized by a textual description (allowing to precise more information about the constraint), an equation, an event, a priority and a depth. The equation corresponds to an invariant property of the Model, which must hold during the whole design phase. If one or more events are specified in a constraint, the associated equation is checked whenever the events are produced by the GME environment. Examples of events are *On Create*, *On Delete* and *On Connect*. When no event is specified, the constraint is only checked on user demand. The priority specified on a constraint is an integer: from 1 (for the highest) to 10 (for the lowest). Finally, events can be produced by the constrained FCO itself or from its descendant FCOs (e.g. this is the case for Models, which may include other FCOs). The depth characterizes the sensitiveness of the constraint: 0 for events from the directly concerned FCO, 1 for events from the FCO or any of its direct descendant and *Any* for events from any of its descendants.

Constraint: CheckParameterConnections

Description: A *Parameter* can only be affected by a constant expression (numeric, boolean, merge and extraction on constant expression, or a *TypeDeclaration* or a *ModelDeclaration*).			
Attach to: *ModelsWithDataflow*	Event: on Close Model	Priority : 2	Depth : 0

Constraint: CheckPragma

Description: To create a new pragma, select first NEW_PRAGMA in the Polychrony Pragma field.			
Attach to: *Pragma*	Event: on Change Attribute	Priority : 2	Depth : 0

Constraint: ConnectedOnlyIfModelType

Description: A *TypeDeclaration* can only be connected as source to a *Parameter* if it is a model.			
Attach to: *TypeDeclaration*	Event: on Connect and on Change Attribute	Priority : 2	Depth : 0

Constraint: CyclicReference

Description: A Connection cannot have the same object at both extremity.			
Attach to: *Connections*	Event: on Create	Priority : 1	Depth : 0

Constraint: DefAndPDefConstraint

Description: A signal and all its references must be defined by a definition connection or by several partial definition connections. An input (and not its reference) can only be connected by one Definition connection.			
Attach to: *SignalOrRef*	Event: on User Demand only	Priority : 2	Depth : 0

Constraint: EmptyAssertion

Description: The assertion expression is empty.			
Attach to: *Assertion*	Event: on User Demand only	Priority : 2	Depth : 0

Constraint: IfThenElseComplete

Description: *IfThenElse* expressions must have one Then connection, one Else connection and at least one condition expression.			
Attach to: *ModelsWithDataflow*	Event: on Close Model	Priority : 2	Depth : 0

Constraint: IndexPriorities

Description: *Index* can only have *Input*s with priority value equal to 1, 2 and/or 3 to represent the affectation to the *From*, *To*, and/or *Step* field.			
Attach to: *OrderedInputs*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: NoClockOfConnectionForBoolOrNum

Description: When the destination FCO is a *BoolExpr* or a *NumExpr*, do not use the signal clock or the source field attributes. Specify the use of signal clock or the source signal inside the *BoolExpr* or a *NumExpr* expression.			
Attach to: *ExpressionInputs* and *FirstOperande*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: NoDstFieldForInputDefinition

Description: No destination field for *Definition* or *PartialDefinition* whose destination is an *Input*. Use *Tuple* to affect multiple field signals.			
Attach to: *Input*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: NoStatevarInTypeDeclaration

Description: No *Local* Atom with *Statevar* attribute set to true in *TypeDeclaration*.			
Attach to: *TypeDeclaration*	Event: on New Child and on Change Attribute	Priority : 1	Depth : 1

Constraint: OnlyClockConstraintInBundle

Description: You can only put clock constraints and clock relations in bundle type *TypeDeclaration*.			
Attach to: *TypeDeclaration*	Event: on New Child and on Change Attribute	Priority : 1	Depth : 0

Constraint: OnlyDstConnForUpLvl

Description: Connections to *Output*s can only be created in the model where *output*s are defined.			
Attach to: *Output*	Event: on Connect	Priority : 1	Depth : 0

Constraint: OnlyDstConnForUpLvlOfModelInstance

Description: *Input* can be only connected as destination from the upper level of the *ModelInstance*. You cannot connect to input of *ModelDeclaration*.			
Attach to: *Input*	Event: on Connect	Priority : 1	Depth : 0

Constraint: OnlyIndexOrIdentifier

Description: You can only connect Identifiers or *Index* expressions.			
Attach to: *Iterate*	Event: on Connect	Priority : 1	Depth : 0

Constraint: OnlyInOutForModelProcess

Description: You can only put *Input*, *Output*, *Parameter* and *Specifications* in Model process (action, function, node or process).			
Attach to: *TypeDeclaration*	Event: on New Child and on Change Attribute	Priority : 1	Depth : 0

Constraint: OnlyLocalInBundleOrStruct

Description: You can only add *Local* signals in 'struct' or 'bundle' type.			
Attach to: *TypeDeclaration*	Event: on New Child and on Change Attribute	Priority : 1	Depth : 0

Constraint: OnlyOneDefaultValue

Description: A signal can only have one default value.			
Attach to: *PartialDefinition*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: OnlyOneElseCaseAsDst

Description: Only one else case can be connected to a *SwitchCase* Atom.			
Attach to: *SwitchCase*	Event: on Connect and on Change Attribute	Priority : 1	Depth : 0

Constraint: OnlyOneExpressionInputs

Description: This expression can only have one *ExpressionInputs* Connection.			
Attach to: *ArrayRestructuration*, *Counter*, *Repetition*, *Delay*, *Window*, and *Not*	Event: on Connect	Priority : 1	Depth : 0

Constraint: OnlyOneIdentifierAsInput

Description: You can only connect one identifier (signal or constant or reference) as source input and only Cases as destination outputs.			
Attach to: *SwitchCase*	Event: on Connect	Priority : 1	Depth : 0

Constraint: OnlyOrderedInOutIfParameter

Description: A *ModelInstance* can have *OrderedInput/OrderedOutput/OrderedParameter* Connections only if it refers to a *Parameter* of type action/function/node/process.			
Attach to: *ModelInstance*	Event: on Connect and on Change Association	Priority : 2	Depth : 0

Constraint: PositiveDelay

Description: The delay must be greater or equal to 1.			
Attach to: *Delay*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: PositiveRepetitionNumber

Description: The number of repetition must be greater or equal to 1.			
Attach to: *Repetition*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: PositiveSizeOfWindow

Description: The size of the window must be greater or equal to 1.			
Attach to: *Window*	Event: on Change Attribute	Priority : 1	Depth : 0

Constraint: PrivateOnlyInModule

Description: The private status can only be used in *Module*.			
Attach to: *InterfaceDefinition*, and *ConstantValue*	Event: on Change Attribute	Priority : 3	Depth : 0

Constraint: ReferOnlyModelProcess

Description: *ModelInstance* cannot refer to a *Parameter* which is neither an action, neither a function, neither a node, nor a process.			
Attach to: *ModelInstance*	Event: on Change Association	Priority : 1	Depth : 0

Constraint: UniqueName

Description: The name of each entity in a Model must be unique.			
Attach to: *ModelsWithClockRelations* and *Module*	Event: on Close Model	Priority : 1	Depth : 0

Constraint: UniquePriorityOrderedConnections

Description: All priorities must be different for Connections whose destination is a *Merging*, a *SequentialDefinition*, a *CartesianProduct*, a *ModelInstance*, an *Index*, a *Concatenation*, an *ArrayEnumeration*, a *Tuple*, or an *Iterate*.			
Attach to: *ModelsWithDataflow*	Event: on Close Model	Priority : 2	Depth : 0

Constraint: ValidModelInstance

Description: The parent of the referenced *ModelDeclaration* or *Parameter* must be an ancestor of the *ModelInstance*.			
Attach to: *ModelsWithDataflow*	Event: on New Child and on Change Association	Priority : 1	Depth : 1

Constraint: ValidName

Description: Format of the name is invalid. The name must begin with an alphabetic character and be completed with alphanumeric characters and/or underscore.			
Attach to: *ModelsWithClockRelations*, *ModelInstance*, *Module*, *Identifiers*, and *Connections*	Event: on Change Property	Priority : 1	Depth : 0

Constraint: ValidRef

Description: The parent of the referenced signal (or constant) must be a strict ancestor (cannot have the same direct parent) of the reference.			
Attach to: *ModelsWithDataflow*	Event: on New Child and on Change Association	Priority : 1	Depth : 1